Chapter

Medical Product Line Architectures

12 years of experience

B.J. Pronk
Philips Medical Systems

Key words: Example architectures, product line architectures, styles and patterns

Abstract: The product line architectures for diagnostic imaging equipment like CT, MRI and conventional X-Ray have to cope with large variations (in hardware and application functions) combined with a high level of integration between their embedded applications. Therefore a primary goal of these architectures is to avoid monolithic applications while retaining the required integrated behaviour. Furthermore, an easy and independent variation of the constituting components is essential. The product line architecture described in this paper gives one recent example solution to this problem. This example presents a layered, event driven; resource restricted system based on the model view controller pattern. Its technical implementation relies heavily on state of the art desktop (WindowsNT™) and component techniques (DCOM). For this architecture orthogonality and (binary) variation have been the key design goals. Several views on this architecture, the conceptual, technical and process models are discussed. In all three views the rationale of the chosen concepts and their relation to the problems indicated above is shown.

1. MEDICAL ARCHITECTURES

1.1 Introduction

Philips Medical Systems is one of the worlds leading suppliers of diagnostic imaging equipment. Its product range includes conventional X-ray, Computed Tomography (CT), Magnetic Resonance Imaging (MRI) and Ultra Sound (US) equipment. These product families, usually called modalities, come in many variants of which only small quantities (100-1000) are being produced, enforcing reuse of development effort and product family architectures for all of them. In this paper the main issues encountered in the architecture development of these product families will be discussed. For illustration a recent example architecture will be presented.

1.2 Characteristics of medical software environment

The main characteristics of the Philips Medical embedded software development environment are:

- Distributed, multi processor
- Real time embedded and standard desktop environments
- Large amount of code (> 10^6 lines of code) per system
- Large software engineering groups (> 100 FTE's)
- Software is by far the fastest growing component of all products
- Long product support, maintenance and extensions (10-15 year).
- Long running projects (2-3 years)
- Distributed development
- Small product series (< 1000/year)
- Strict quality, legal and safety requirements

1.3 Architecture Overview

From a physical viewpoint most of the products mentioned above are constructed along the same principles. They are centred around a host processor, running a desk top operating system, that controls a set of modality specific peripheral devices which are needed to generate, process and view images. These peripherals are normally large, expensive and controlled locally by embedded real time processors or DSP's. Examples are high-tension amplifiers, patient support mechanics, RF-coils etc. The set of peripherals is unique to a single product family, although many variations of individual peripherals are usually supported within one product family.

On the host of all modalities, similar software applications linked with the user workflow can be identified:

- Database and patient administration for entering patient data in the system
- Acquisition, that programs all devices for image generation,
- Viewing application that allows the user to review and process the acquired images,
- Image handling applications that support all further handling of the information obtained during the examination such as printing, archiving and network communication.

In the following *Figure 1* this architecture is sketched:

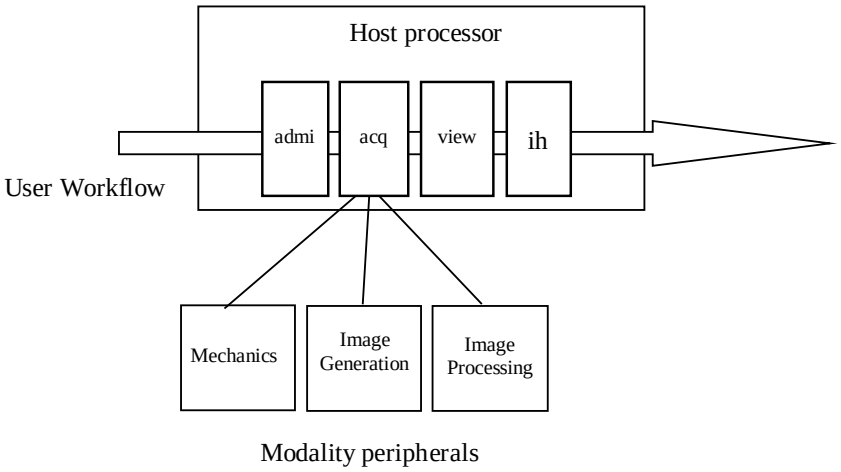

Figure 1. Medical Architecture Overview

2. MAIN ARCHITECTURAL ISSUES

The main issues to be addressed by the software architecture of medical product families can be summarised as:

- Reuse: The need to support many different product family members, serving a variety of application areas and operating in many different (hardware) configurations, with one shared code base.
- Independence: Allowing parallel, independent and incremental development for specific family members.
- Time to market: Allow efficient addition of new functionality for the various family members in reaction to changing market needs.

In the remaining of this paragraph the main aspects of these problem areas are explored somewhat further:

- Reuse:
  - Medical products come in many configurations (types of hardware, software options) serving various market segments and application areas. Yet within one product family (X-Ray, MRI etc.) a lot of functionality can be identified that is common to all family members. Because of the long lifetime, small production numbers and enormous code base (investments) of most product families these variations must be handled by the configuration of a single basic platform.
- Independence:
  - Often the variations indicated above influence significant properties of the system (e.g. maximum frame speed), that propagate throughout the entire architecture. As a consequence of this current implementations show cross-dependencies throughout the entire software system. Other symptoms of these phenomena are multiple definitions and extensive and complex branches.
  - Furthermore the current practice of source code reuse introduces heavy compile time dependencies between all components. Independent development and delivery are virtually impossible in this situation. Furthermore this strong coupling requires extensive testing at every change yielding ever-longer test cycles.
  - The software applications of a medical device show a very integrated behaviour to their users. This is reflected in software dependencies on all levels (user interface, application and technical level). Examples of these are the sharing of the current patient and image between applications, the use of shared (hardware) resources and the compensation of imperfections of one device in another one.
- Time to market
  - Many new features, acquisition techniques and hardware devices are added to medical products over the lifetime of the software architecture. These extensions are often accompanied by extensive growth of the coupling in the system since the necessary interfaces do not exist in the architecture. Continuous engineering by an ever-varying population of developers, forgetting or even unaware of the original architecture further aggravates this situation.
  - Medical devices contain a lot of persistent data; patient and image related data, system settings, and configuration of the system and its components and calibration data. Each of these settings depends on the software level of the components, the actual available hardware and

the configuration and options available on a system. This strongly coupled set of data imposes a significant barrier to change. The same goes for exchange of data between different releases, systems and off-line tools that introduce many compatibility problems.

– Dedicated solutions and proprietary techniques have been widespread throughout the professional industry. In view of the advance of modern desktop operating systems with their myriad of applications, productivity tools and high innovation rate this legacy has become one of the sources of low innovation speed.

3. AN EXAMPLE SOLUTION

3.1 Introduction

In this paragraph a recent example of medical product family architecture is described. In its quest for a solution to the three main architectural issues introduced in the previous paragraph it applies the following principles.

1. Avoiding a monolithic design by de-coupling and localisation. Every component can be replaced in isolation.
2. Binary reuse of components, reducing compile time dependencies.
3. Use of standard technology and tools for productivity enhancement
4. Division of the product family development in a generic (platform) part and member specific parts. Addition of specific parts should be possible in independent parallel activities.

None of the principles stated above is very revolutionary. And of them only binary reuse of components can be considered to be relatively new, since enabling technology has recently become widely available (Com, Corba). Yet we think that the strict adherence to these principles and the actual implementation followed has lead to a system coping with the main architecture issues better then any of our previous implementations. This new product family architecture has been modelled in several views, which will be described in detail in the remaining of this paper:

– The conceptual architecture view: Describing the solutions and rules as applied to tackle the main architectural issues of decomposition vs. co-operation. The actual design of the system is done according to these

solutions. This view will receive most attention in the discussion in this paper.

– The technical architecture view: This view describes all additional constructs necessary on top of the conceptual view (e.g. I/O classes, caching mechanisms) to realise the system. It also describes the hardware (processors, busses etc.) and software (Operating system, protocols etc.) infrastructure and technology choices.

Furthermore we will describe the process architecture. However this is no view on the same level as the previous two. In fact both within the technical and conceptual architecture a process architecture view can be identified. Within the conceptual architecture this describes the general approach for handling the required (application) concurrence. Within the technical architecture it describes the deployment of the elements of the decomposition to threads, processors and processes. This latter point will not be addressed in this paper.

The architecture is thus described by: a set of rules and concepts, a series of technology and infrastructure choices, the decomposition of the solution domain into so called Units, their deployment to the infrastructure and the set of interfaces between them. As much as possible the rules and concepts are expressed in formal terms, to allow automatic verification of adherence to them in both the platform and specific developments.

The presented three models (conceptual, technical, process) resemble quite closely three of the views as expressed in [P.B. Kruchten]. On a lower level the same views are used in the design of the individual Units that fit into this architecture. This set of views has been selected since they have proven to be sufficient input for the designers of these Units to complete their requirements and designs in relative independence.

3.2 Conceptual architecture

3.2.1 Layering

The conceptual architecture of the product family describes the concepts, rules and tactics that implement the principles described in paragraph 3.2. Note that the conceptual architecture mainly addresses principle 1 (the localisation and de-coupling). Note also that the conceptual architecture is almost independent of the underlying technology, which is added only at a later stage.

The product family architecture has the following main concepts:

– Layering: The decomposition of the system in a number of (independent) abstraction layers, from the bottom up:

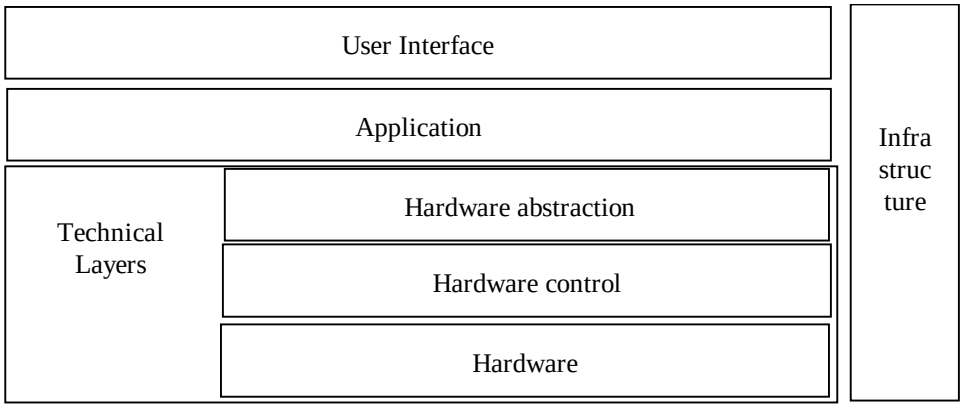

Figure 2. Layered set-up of product family architecture

– Technical layer, consisting of the following sub layers:
  – Hardware: Basic digital and analogue hardware and their controllers.
  – Hardware control: Drivers and real time control of hardware, shielding the low level details of the hardware implementation like registers, addresses, interrupts etc.
  – Hardware abstraction: Abstraction layer offering a domain specific abstraction of the underlying type of hardware (e.g. the X-Ray generation part in a CT-family).
– Application layer: The actual user functions realised with this equipment.
– User interface layer: The presentation layer, taking care of display and user interaction.

Next to these three layers there is an infrastructure layer that is used by all. The three layers and their sub-layers supply a true abstraction; i.e. they are not transparent to the layers above them. Each of these layers can therefore be replaced independently of the surrounding layers. This is one of the major features supporting the variation within the product family. Examples of this are:

- Different user interfaces for the members of the family
- Various implementations of the geometry part of an X-Ray system
- Implementing function from several application areas on top of the common (domain) abstraction layer. E.g. a cardio and a neurological MRI application.

3.2.2 Conceptual building blocks

Within each layer several independent Units are distinguished, which should not interact with each other. Therefore each of these Units is as much as possible self-contained. Local are e.g. stored in the Unit itself. The conceptual building blocks used within the three layers are:

- Services: A main structuring element of the architecture is the service concept. A service is a software entity that autonomously executes a number of tasks for another part of the software, guarding a set of resources. A service is a completely isolated part of the architecture that also keeping its own configuration etc. The technical layer consists of a set of these services, one for each device. These are in an X-ray system e.g. the services for the generator and detector and in a MRI system the services for the gradient amplifier and the RF coils.
- Applications: A number of applications such as reviewing, acquisition, patient and beam positioning etc. These applications are themselves again services offering an interface to the user interface layer. Applications offer a very uniform interface consisting of commands (in fact the use cases as defined in the functional specifications) and a so-called UI-model that represents all information (data and state) necessary for the user interface.
- User interfaces: The user interface is completely de-coupled of the applications and interacts with the applications through the application service interfaces described above. Through out the system a model, view controller pattern is applied with the user interface being the “view”. The application in fact contains the model (the UI-model) and the controller (the commands). The grouping and appearance of the user interface is not known by the applications. There might be one integrated UI for multiple applications or a single user interface per application.

3.2.3 Independence

The previous steps represent a major step forward in de-coupling the various Units of the system. However interaction between Units can not be avoided completely because of the integrated behaviour aspects described before. Yet we maintain the rule that applications and services of the system will not interact directly with each other. This will be supported by the following mechanisms.

- Event driven: Another main concept of the architecture that supports de-coupling is notification. Objects in this architecture may issue events (notifications) that can be received (if requested) by so called observers. This mechanism works both in process and over process boundaries. The source of the event in this mechanism is not aware of its observers. All upward communication between Units is based upon this mechanism.
- Integration: All system wide known data (e.g. patient data but also currencies) is stored in a separate service called integration and datamodel. Applications never directly exchange data such as a change in the current patient. Instead the current patient object in the integration service is updated by the patient administration application. All interested applications may be notified about this change through the notification mechanism described in the previous point. The integration service is closely linked to the database since a lot of this information is also stored persistently.
- Automation: Many sequences of operations in the system are pre-programmed. After an acquisition, the system goes over to reviewing mode; data are forwarded to an archive etc. After closing an examination, data are forwarded to printers, the Radiology Information System etc. Such functionality is located in a separate automation service that receives completion notifications from the applications and starts the relevant actions. Again the applications do not interact directly.
- All system characteristics are derived from the available resources an application can get from the services it uses. There is no hard coding in the applications of restrictions of the underlying services. This implies that addition of resources will increase the capabilities of the system without additional coding.

3.2.4 Process architecture

This paragraph describes the concepts used for decomposing the system in separate concurrent processes starting with the application requirements for concurrence. From a user point of view the system should deliver the following levels of concurrence:

- Multiple users operating separate applications concurrently. This will be handled by defining all applications to be separate processes.
- All long running non interactive user functions (e.g. printing, export, archiving) have to be performed as background parallel processes since the user wants to be free to do other actions while these functions are executed.
- Long running interactive user functions (like screen build up) have to be performed in parallel processes to retain user interface responsiveness. For these functions the user should be able to cancel or overrule it.

From a technical point of view additional concurrence is introduced in the system since asynchronous hardware has to be controlled. So all services handling hardware have to be separate conceptual processes. Yet another technical point of concurrence follows from the services concept itself. Lengthy actions are often distributed over a client and multiple services. A service request may take considerable time to complete since often handling of hardware I/O is involved. During the time that the service request is handled the application can often do other useful things (e.g. starting other service requests in parallel). It is a matter of choice where to put the conceptual processes for handling lengthy service requests. We choose to put this in the services itself. So all lengthy service requests have to run in separate conceptual processes. This also implies that these service requests will complete asynchronously (and use notification to signal completion).

From this initial selection yet more concurrence requirements can be derived. Since multiple conceptual processes are active in parallel, shared resources (e.g. database, context) are introduced. Therefore additional conceptual processes will be introduced to serialise access to these shared resources.

3.3 Technical Architecture

3.3.1 Introduction

The technical architecture of the system supports especially the following principles from paragraph 3.1 :

- Binary reuse of components.
- Enhance productivity by application of standard, state of the art technology
- Building a generic platform with product specific additions.

3.3.2 Use of standard environment

Professional industrial environments have long worked with proprietary solutions. However the advance of standard desktop environments, market pressure and the need for productivity increase drives the industry towards usage of standard solutions and open standards. Note that this is not only a matter of money. Even where money is no argument, the time and people needed to create from scratch something competing with standard desktop environments forms a tremendous bottleneck. Finally the innovation rate of desktop environments is now so high that proprietary solutions will probably be outdated before they are introduced. Therefore the following approach is chosen for the new product family architecture:

- Allocate as much as possible functionality in software on custom hardware components. Only build dedicated hardware when processing/responsiveness can not (cost effectively) be delivered by such a platform.
- Allocate as much as possible functionality in a standard desktop environment, only use a real time operating system environment when strictly required (for performance, safety or graceful degradation).
- Use of standard PC-architecture and technology as much as possible (PCI, Intel x86, WindowsNT, Microsoft Foundation Classes, Windows User interface, WindowsNT services etc.).
- Use of standard software packages (database, license management, network)

– Use of internet technology (Java/HTML/Browser, Windows-NT peer web server) for (remote) service.

3.3.3 Binary exchange

Classical reuse programs are often based on source code level reuse. This approach introduces strong compile time dependencies. Furthermore it does not support true reuse since extensive testing is still required in the new code/compile environment. This situation is even further aggravated when using object oriented languages and deep inheritance trees. Based on these experiences it has been decided that the new product family architecture will be based on binary variation. The following choices have been made in this area:

– Component based development (DCOM) based on binary exchange, allowing flexible allocation of UI, application and services.
– All interfaces in the system will be expressed in IDL, DCOM will be used for all communication between Units.
– All notification between Units will be based on the COM connection point mechanism.
– DCOM however is only used as an interface mechanism, all implementation classes are strictly separated from this interface shell.
– Interfaces are considered immutable even when extending e.g. ranges of enumerated types or error codes, new interface versions will be introduced.
– Apply component technology to define frameworks for all extensible parts of the system. A framework consists of a set of interfaces and some generic functionality. E.g. the acquisition application is a framework in which (binary) components can be added to support additional acquisition procedures.

4. CONCLUSIONS

Medical equipment architecture has to focus on orthogonality and independence to support a viable product family concept. The rigorous pursued de-coupling in the presented product family architecture allows for the development of completely localised and highly independent components. The use of DCOM as standard interface technology enables versioning, strict interface management and the delivery of components that

are thoroughly tested. In addition applying standard technology and components will reduce time to market significantly. This combined approach has resulted in a generic platform, which through addition of system specific components can be specialised in parallel developments.

REFERENCES:

"The 4+1 View Model of Architecture ", P.B. Kruchten, IEEE Software, November 1995, pp. 42-50